This comprehensive tutorial covers all the essential aspects of Python's `contextlib` module with detailed explanations of when and where to use each feature. The key takeaways are:

Summary of When and Where to Use Each Feature

@contextmanager

- **When**: Custom setup/cleanup logic, temporary state changes, reusable patterns
- **Where**: Utility modules, test fixtures, database transactions, resource management
- **Real-world**: Database connections, file operations with logging, performance monitoring

ExitStack

- **When**: Variable number of resources, runtime-determined resources, partial failures
- **Where**: Batch processing, resource pools, testing frameworks, data pipelines
- **Real-world**: Processing multiple files, scaling worker processes, ETL pipelines

suppress

- **When**: Expected exceptions you want to ignore, optional operations, cleanup
- **Where**: Cleanup scripts, optional features, graceful degradation
- **Real-world**: File deletion, optional service calls, plugin cleanup

redirect_stdout/stderr

- **When**: Capturing output from unmodifiable functions, testing, quiet modes
- **Where**: Testing frameworks, legacy code integration, monitoring systems
- **Real-world**: Testing print-based functions, capturing subprocess output, logging

nullcontext

- **When**: Conditional context management, optional features, testing
- **Where**: Configuration-driven apps, feature flags, testing scenarios
- **Real-world**: Optional transactions, conditional compression, testing different modes

closing

- **When**: Objects with close() method that aren't context managers, thir## closing {#closing}

The `closing` context manager ensures that objects with a `close()` method are properly closed.

When to use:

- When working with objects that have a `close()` method but aren't context managers
- For network connections, file-like objects, or custom resources
- When integrating with third-party libraries that don't support context managers
- For ensuring resource cleanup in legacy code

Where to use:

- Network programming (sockets, HTTP connections)
- File-like objects from third-party libraries
- Custom resource classes
- Database connections that aren't context managers

Basic Usage

```python
from contextlib import closing
import urllib.request

def fetch_web_data(url):
    """
    USE CASE: HTTP request with# Complete contextlib Tutorial

## Table of Contents
1. [Introduction to Context Managers](#introduction)
2. [The contextlib Module](#contextlib-module)
3. [contextmanager Decorator](#contextmanager-decorator)
4. [ExitStack](#exitstack)
5. [suppress](#suppress)
6. [redirect_stdout and redirect_stderr](#redirect)
7. [nullcontext](#nullcontext)
8. [closing](#closing)
9. [Advanced Patterns](#advanced-patterns)
10. [Best Practices](#best-practices)

## Introduction to Context Managers {#introduction}

Context managers implement the context management protocol through `__enter__` and `__exit__` methods. The `with` s

### Basic Context Manager Example

```python
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        print(f"Opening file: {self.filename}")
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"Closing file: {self.filename}")
        if self.file:
            self.file.close()
        # Return False to propagate exceptions
        return False

# Usage
```
```

```python
# Usage
with FileManager('example.txt', 'w') as f:
    f.write('Hello, World!')
```

The contextlib Module {#contextlib-module}

The `contextlib` module provides utilities to create context managers more easily than defining classes with `__enter__` and `__exit__` methods.

```python
import contextlib
import os
import sys
from contextlib import contextmanager, ExitStack, suppress
```

contextmanager Decorator {#contextmanager-decorator}

The `@contextmanager` decorator transforms a generator function into a context manager.

When to use:

- When you need custom setup/cleanup logic that's too complex for built-in context managers
- When creating reusable resource management patterns
- When you want to avoid writing full context manager classes
- For temporary state changes that need guaranteed restoration

Where to use:

- In utility modules for common resource patterns
- In test fixtures for setup/teardown
- In configuration management
- In database transaction handling

Basic Usage

python

```python
@contextmanager
def file_manager(filename, mode):
    """
    USE CASE: Custom file handling with logging
    WHEN: You need to track file operations or add custom behavior
    WHERE: Utility modules, logging systems, audit trails
    """
    print(f"Opening file: {filename}")
    try:
        f = open(filename, mode)
        yield f
    finally:
        print(f"Closing file: {filename}")
        f.close()

# Usage
with file_manager('example.txt', 'w') as f:
    f.write('Hello from contextmanager!')
```

Database Connection Example

python

```python
@contextmanager
def database_connection(db_name):
    """
    USE CASE: Database transaction management
    WHEN: You need guaranteed connection cleanup and transaction rollback
    WHERE: Database access layers, ORM implementations, data processing scripts
    REAL-WORLD SCENARIO: E-commerce checkout process where payment failures
                         should rollback inventory changes
    """
    print(f"Connecting to database: {db_name}")
    # Simulate database connection
    connection = f"connection_to_{db_name}"
    try:
        yield connection
    except Exception as e:
        print(f"Database error: {e}")
        # Rollback transaction
        print("Rolling back transaction")
        raise
    finally:
        print(f"Closing connection to: {db_name}")

# Usage in e-commerce scenario
def process_order(order_id):
    with database_connection('orders_db') as conn:
        # These operations will be rolled back if any fail
        update_inventory(conn, order_id)
        process_payment(conn, order_id)
        send_confirmation(conn, order_id)
```

Timer Context Manager

```python
import time

@contextmanager
def timer(name="Operation"):
    """
    USE CASE: Performance monitoring and profiling
    WHEN: You need to measure execution time of code blocks
    WHERE: Performance testing, optimization, debugging slow operations
    REAL-WORLD SCENARIO: API endpoint monitoring, batch job performance tracking
    """
    start_time = time.time()
    print(f"Starting {name}...")
    try:
        yield
    finally:
        end_time = time.time()
        print(f"{name} completed in {end_time - start_time:.2f} seconds")

# Usage in API monitoring
def api_endpoint_handler():
    with timer("User authentication"):
        authenticate_user()

    with timer("Database query"):
        fetch_user_data()

    with timer("Response serialization"):
        serialize_response()
```

Temporary Directory Example

```python
import tempfile
import shutil

@contextmanager
def temporary_directory():
    """
    USE CASE: Temporary file operations with guaranteed cleanup
    WHEN: Processing files that need temporary storage
    WHERE: Image processing, data transformation, testing, build systems
    REAL-WORLD SCENARIO: Video processing pipeline that needs temporary files
                         for intermediate steps
    """
    temp_dir = tempfile.mkdtemp()
    print(f"Created temporary directory: {temp_dir}")
    try:
        yield temp_dir
    finally:
        shutil.rmtree(temp_dir)
        print(f"Cleaned up temporary directory: {temp_dir}")

# Usage in video processing
def process_video(input_file, output_file):
    with temporary_directory() as temp_dir:
        # Extract frames
        frames_dir = os.path.join(temp_dir, 'frames')
        os.makedirs(frames_dir)

        # Process frames (apply filters, etc.)
        processed_dir = os.path.join(temp_dir, 'processed')
        os.makedirs(processed_dir)

        # Reassemble video
        # All temporary files are automatically cleaned up
```

ExitStack {#exitstack}

ExitStack allows you to manage multiple context managers dynamically.

When to use:

- When you need to manage a variable number of resources
- When resources are determined at runtime
- When you want to collect multiple context managers
- When you need to handle partial failures in resource acquisition

Where to use:

- File processing systems handling multiple files
- Resource pools and connection management
- Testing frameworks managing multiple fixtures
- Batch processing systems

Basic ExitStack Usage

python

```python
from contextlib import ExitStack

def process_multiple_files(filenames):
    """
    USE CASE: Batch file processing with guaranteed cleanup
    WHEN: You need to process multiple files simultaneously
    WHERE: Data processing pipelines, log analyzers, backup systems
    REAL-WORLD SCENARIO: Merging multiple CSV files into a single report
    """
    with ExitStack() as stack:
        files = [stack.enter_context(open(fname, 'r')) for fname in filenames]

        # Process all files - they'll all be closed automatically
        for i, f in enumerate(files):
            print(f"File {i+1} content: {f.read()[:50]}...")

# Usage in data processing
def merge_csv_files(input_files, output_file):
    with ExitStack() as stack:
        # Open all input files
        readers = [stack.enter_context(open(f, 'r')) for f in input_files]
        # Open output file
        writer = stack.enter_context(open(output_file, 'w'))

        # Process all files - guaranteed cleanup even if one fails
        for reader in readers:
            writer.write(reader.read())
```

Dynamic Context Manager Registration

python

```python
@contextmanager
def resource_manager(resource_id):
    print(f"Acquiring resource {resource_id}")
    try:
        yield f"resource_{resource_id}"
    finally:
        print(f"Releasing resource {resource_id}")

def manage_resources(resource_ids):
    """
    USE CASE: Dynamic resource allocation
    WHEN: Resource requirements are determined at runtime
    WHERE: Cloud resource management, worker pools, microservices
    REAL-WORLD SCENARIO: Scaling worker processes based on queue length
    """
    with ExitStack() as stack:
        resources = []
        for res_id in resource_ids:
            resource = stack.enter_context(resource_manager(res_id))
            resources.append(resource)

        print(f"Working with resources: {resources}")
        return resources

# Usage in worker pool management
def scale_workers(current_queue_size):
    needed_workers = min(current_queue_size, MAX_WORKERS)
    worker_ids = list(range(needed_workers))

    workers = manage_resources(worker_ids)
    # Process queue with available workers
    # All workers are cleaned up automatically
```

Exception Handling with ExitStack

```python
def safe_file_operations(filenames):
    """
    USE CASE: Robust file processing with partial failures
    WHEN: Some files might be missing or inaccessible
    WHERE: Log processing, backup systems, data migration
    REAL-WORLD SCENARIO: Processing log files where some might be rotated or deleted
    """
    with ExitStack() as stack:
        files = []
        for filename in filenames:
            try:
                f = stack.enter_context(open(filename, 'r'))
                files.append(f)
            except FileNotFoundError:
                print(f"Warning: {filename} not found, skipping")

        # Process only the files that opened successfully
        for f in files:
            print(f"Processing: {f.name}")
            # Process file content

# Usage in log analysis
def analyze_daily_logs(date_range):
    log_files = [f"/var/log/app-{date}.log" for date in date_range]
    safe_file_operations(log_files)  # Handles missing logs gracefully
```

suppress {#suppress}

The `suppress` context manager suppresses specified exceptions.

When to use:

- When you expect certain exceptions and want to ignore them
- For cleanup operations that might fail
- When implementing optional functionality
- For defensive programming against known edge cases

Where to use:

- File cleanup operations
- Optional feature implementations
- Graceful degradation scenarios
- Testing and development environments

When NOT to use:

- Never suppress exceptions you don't understand
- Avoid suppressing broad exception types
- Don't use for control flow (use try/except instead)

Basic Usage

```python
from contextlib import suppress

# Suppress specific exceptions
"""
USE CASE: Safe file cleanup
WHEN: You want to delete files that might not exist
WHERE: Cleanup scripts, temporary file management, deployment scripts
REAL-WORLD SCENARIO: Cleaning up after a failed deployment
"""
with suppress(FileNotFoundError):
    os.remove('nonexistent_file.txt')
    print("This won't print if file doesn't exist")

print("This will always print")
```

Multiple Exception Types

python

```python
def safe_data_conversion(data_dict):
    """
    USE CASE: Data conversion with graceful degradation
    WHEN: Processing user input or external data that might be malformed
    WHERE: API endpoints, data import systems, configuration parsers
    REAL-WORLD SCENARIO: Processing CSV data with inconsistent formats
    """
    results = {}

    for key, value in data_dict.items():
        with suppress(ValueError, TypeError, KeyError):
            # Try to convert to integer
            results[key] = int(value)
            continue

        with suppress(ValueError, TypeError):
            # Try to convert to float
            results[key] = float(value)
            continue

        # Keep as string if conversion fails
        results[key] = str(value)

    return results

# Usage in data processing
csv_row = {'id': '123', 'price': '45.67', 'name': 'Widget', 'invalid': 'N/A'}
converted = safe_data_conversion(csv_row)
```

Practical Example: Cleanup Operations

```python
def cleanup_files(filenames):
    """
    USE CASE: Robust cleanup operations
    WHEN: You need to clean up files that might not exist or be locked
    WHERE: Build systems, deployment scripts, test cleanup, cache clearing
    REAL-WORLD SCENARIO: Cleaning up after a test suite run
    """
    for filename in filenames:
        with suppress(FileNotFoundError, PermissionError):
            os.remove(filename)
            print(f"Removed: {filename}")

# Usage in test cleanup
def cleanup_test_artifacts():
    test_files = [
        'test_output.txt',
        'temp_data.json',
        'cached_results.pkl',
        'debug.log'
    ]
    cleanup_files(test_files)  # Won't fail if files don't exist
```

redirect_stdout and redirect_stderr {#redirect}

These context managers redirect standard output and error streams.

When to use:

- When you need to capture output from functions you can't modify
- For testing functions that print to stdout/stderr
- When integrating with legacy code that uses print statements
- For creating quiet modes in applications

Where to use:

- Testing frameworks
- Command-line tools with quiet modes
- Log processing and analysis
- Legacy code integration

Redirecting stdout

python

```python
from contextlib import redirect_stdout
import io

def capture_output_example():
    """
    USE CASE: Testing functions that print output
    WHEN: You need to verify what a function prints
    WHERE: Unit tests, output validation, debugging
    REAL-WORLD SCENARIO: Testing a report generation function
    """
    output_buffer = io.StringIO()
    with redirect_stdout(output_buffer):
        print("This goes to the buffer")
        print("So does this")

    captured_output = output_buffer.getvalue()
    print(f"Captured: {captured_output}")

# Usage in testing
def test_report_generation():
    output = io.StringIO()
    with redirect_stdout(output):
        generate_monthly_report()  # Function that prints report

    report_text = output.getvalue()
    assert "Monthly Summary" in report_text
    assert "Total Sales:" in report_text
```

Redirecting to File

```python
def log_application_output(log_file):
    """
    USE CASE: Logging application output to files
    WHEN: You want to capture all output for debugging or auditing
    WHERE: Production systems, debugging, audit logs
    REAL-WORLD SCENARIO: Batch job that needs to log all output
    """
    with open(log_file, 'w') as f:
        with redirect_stdout(f):
            print("This goes to the file")
            print("Line 2 in file")
            run_batch_process()  # All output goes to file

# Usage in batch processing
def run_nightly_batch():
    log_file = f"/var/log/batch-{datetime.now().strftime('%Y%m%d')}.log"
    log_application_output(log_file)
```

Redirecting stderr

```python
from contextlib import redirect_stderr

def capture_errors():
    """
    USE CASE: Capturing error messages for analysis
    WHEN: You need to programmatically handle error output
    WHERE: Error monitoring, debugging tools, automated testing
    REAL-WORLD SCENARIO: Monitoring third-party library errors
    """
    error_buffer = io.StringIO()
    with redirect_stderr(error_buffer):
        # This would normally go to stderr
        print("This is an error message", file=sys.stderr)

    captured_error = error_buffer.getvalue()
    print(f"Captured error: {captured_error}")

    # Analyze or log the error
    if "error" in captured_error.lower():
        log_error_to_monitoring_system(captured_error)
```

Practical Example: Capturing Function Output

```python
def capture_function_output(func, *args, **kwargs):
    """
    USE CASE: Testing and debugging function output
    WHEN: You need to test functions that print instead of returning values
    WHERE: Legacy code testing, output validation, debugging
    REAL-WORLD SCENARIO: Testing a data processing function that prints progress
    """
    output_buffer = io.StringIO()
    with redirect_stdout(output_buffer):
        result = func(*args, **kwargs)

    return result, output_buffer.getvalue()

def noisy_function(x, y):
    print(f"Processing {x} and {y}")
    print("Doing some work...")
    return x + y

# Usage in testing framework
def test_data_processing():
    result, output = capture_function_output(process_data, large_dataset)

    # Test the result
    assert result.status == 'success'

    # Test the output messages
    assert "Processing complete" in output
    assert "Error" not in output
```

nullcontext {#nullcontext}

`nullcontext` is a context manager that does nothing, useful for conditional context management.

When to use:

- When you need conditional context management based on runtime conditions
- For implementing optional features or modes
- When you want to avoid code duplication between context-managed and non-context-managed code paths
- For testing and mocking scenarios

Where to use:

- Configuration-driven applications
- Optional feature implementations
- Testing frameworks
- Conditional resource management

Basic Usage

python

```python
from contextlib import nullcontext

def process_data(data, use_transaction=False):
    """
    USE CASE: Conditional transaction management
    WHEN: Transactions are only needed in certain scenarios
    WHERE: Database operations, batch processing, configuration-driven apps
    REAL-WORLD SCENARIO: Data processing that optionally uses transactions
                         based on data volume or user settings
    """
    # Conditionally use a context manager
    context = database_connection('mydb') if use_transaction else nullcontext()

    with context as conn:
        # Process data - same code works with or without transaction
        print(f"Processing data with connection: {conn}")
        if use_transaction:
            execute_sql(conn, "BEGIN TRANSACTION")

        process_records(data)

        if use_transaction:
            execute_sql(conn, "COMMIT")

# Usage
process_data("large_dataset", use_transaction=True)   # Uses DB transaction
process_data("small_dataset", use_transaction=False)  # No transaction overhead
```

File Processing Example

```python
def process_file(filename, compress=False):
    """
    USE CASE: Conditional file compression
    WHEN: You want to optionally compress output based on file size or settings
    WHERE: Log processing, backup systems, data archiving
    REAL-WORLD SCENARIO: Backup system that compresses large files but not small ones
    """
    import gzip

    # Conditionally use compression
    context = gzip.open(filename, 'wt') if compress else open(filename, 'w')

    with context as f:
        f.write("This is the content")
        f.write("\nSecond line")
        write_data_to_file(f)  # Same code works for both compressed and uncompressed

# Usage in backup system
def backup_file(source, dest, file_size):
    # Compress files larger than 1MB
    should_compress = file_size > 1024 * 1024
    dest_file = dest + '.gz' if should_compress else dest

    process_file(dest_file, compress=should_compress)
```

closing {#closing}

The `closing` context manager ensures that objects with a `close()` method are properly closed.

When to use:

- When working with objects that have a `close()` method but aren't context managers
- For network connections, file-like objects, or custom resources
- When integrating with third-party libraries that don't support context managers
- For ensuring resource cleanup in legacy code

Where to use:

- Network programming (sockets, HTTP connections)
- File-like objects from third-party libraries
- Custom resource classes
- Database connections that aren't context managers

Basic Usage

python

```python
from contextlib import closing
import urllib.request

def fetch_web_data(url):
    """
    USE CASE: HTTP request with guaranteed connection cleanup
    WHEN: Making HTTP requests where connection might not be automatically closed
    WHERE: Web scraping, API clients, data fetching services
    REAL-WORLD SCENARIO: Fetching data from multiple APIs in a batch job
    """
    with closing(urllib.request.urlopen(url)) as response:
        data = response.read()
        return data

# Usage in web scraping
def scrape_multiple_pages(urls):
    results = []
    for url in urls:
        try:
            data = fetch_web_data(url)
            results.append(process_html(data))
        except Exception as e:
            print(f"Failed to fetch {url}: {e}")
    return results
```

Custom Object Example

python

```python
class CustomResource:
    """
    USE CASE: Custom resource that needs explicit cleanup
    WHEN: You have objects that manage resources but aren't context managers
    WHERE: Hardware interfaces, custom protocols, resource pools
    REAL-WORLD SCENARIO: Managing hardware devices or network connections
    """
    def __init__(self, name):
        self.name = name
        self.is_open = True
        print(f"Opened resource: {name}")

    def close(self):
        if self.is_open:
            print(f"Closing resource: {self.name}")
            self.is_open = False

    def do_work(self):
        if self.is_open:
            print(f"Working with {self.name}")

# Usage with closing
def manage_hardware_device(device_name):
    with closing(CustomResource(device_name)) as resource:
        resource.do_work()
        # Resource is automatically closed even if exception occurs

# Usage in IoT device management
def collect_sensor_data(sensor_configs):
    data = []
    for config in sensor_configs:
        with closing(SensorDevice(config)) as sensor:
            reading = sensor.read_data()
            data.append(reading)
    return data  # All sensors are properly closed
```

Advanced Patterns {#advanced-patterns}

Nested Context Managers

python

```python
@contextmanager
def nested_context(name, depth=0):
    """
    USE CASE: Hierarchical resource management
    WHEN: You need nested scopes with different resource levels
    WHERE: Transaction management, nested locks, hierarchical operations
    REAL-WORLD SCENARIO: Database operations with nested transactions
    """
    indent = "  " * depth
    print(f"{indent}Entering {name}")
    try:
        yield name
    finally:
        print(f"{indent}Exiting {name}")

# Usage in database transaction hierarchy
def process_complex_operation():
    with nested_context("Main Transaction"):
        with nested_context("User Updates", 1):
            update_user_profile()
            with nested_context("Audit Log", 2):
                log_user_changes()

        with nested_context("Notification", 1):
            send_update_notification()
```

Parameterized Context Managers

python

```python
@contextmanager
def performance_monitor(threshold_seconds=1.0):
    """
    USE CASE: Configurable performance monitoring
    WHEN: You need different performance thresholds for different operations
    WHERE: API monitoring, batch processing, performance testing
    REAL-WORLD SCENARIO: Monitoring different types of operations with different SLAs
    """
    start_time = time.time()
    try:
        yield
    finally:
        elapsed = time.time() - start_time
        if elapsed > threshold_seconds:
            print(f"WARNING: Operation took {elapsed:.2f} seconds")
        else:
            print(f"Operation completed in {elapsed:.2f} seconds")

# Usage in API endpoint monitoring
def api_handler():
    with performance_monitor(0.5):  # API calls should be fast
        process_user_request()

    with performance_monitor(5.0):  # Database operations can be slower
        complex_database_query()

    with performance_monitor(30.0):  # Batch operations can take longer
        generate_large_report()
```

Context Manager with State

python

```python
class StateManager:
    """
    USE CASE: Temporary state changes with guaranteed restoration
    WHEN: You need to temporarily modify application state
    WHERE: Testing, configuration changes, feature flags
    REAL-WORLD SCENARIO: Testing with different configuration settings
    """
    def __init__(self):
        self.state = {}

    @contextmanager
    def set_state(self, **kwargs):
        old_state = self.state.copy()
        self.state.update(kwargs)
        try:
            yield self.state
        finally:
            self.state = old_state

# Usage in testing
def test_feature_with_different_configs():
    app_state = StateManager()
    app_state.state = {'debug': False, 'verbose': False, 'feature_x': False}

    # Test with debug mode
    with app_state.set_state(debug=True) as state:
        result = run_feature_test()
        assert result.debug_info is not None

    # Test with feature flag enabled
    with app_state.set_state(feature_x=True) as state:
        result = run_feature_test()
        assert result.has_feature_x is True

    # Original state is restored
    assert app_state.state['debug'] is False
    assert app_state.state['feature_x'] is False
```

Async Context Managers

python

```python
import asyncio
from contextlib import asynccontextmanager

@asynccontextmanager
async def async_resource():
    """
    USE CASE: Asynchronous resource management
    WHEN: Working with async/await and resources that need async cleanup
    WHERE: Async web frameworks, database connections, async I/O operations
    REAL-WORLD SCENARIO: Managing async database connections in web applications
    """
    print("Acquiring async resource")
    try:
        # Simulate async setup
        await asyncio.sleep(0.1)
        yield "async_resource_handle"
    finally:
        print("Releasing async resource")
        await asyncio.sleep(0.1)

# Usage (in async function)
async def async_web_handler():
    async with async_resource() as resource:
        print(f"Using {resource}")
        # Perform async operations
        await process_async_request()
        await update_async_database()

# Usage in FastAPI or similar async frameworks
async def api_endpoint():
    async with database_connection_pool() as conn:
        async with redis_cache() as cache:
            result = await complex_async_operation(conn, cache)
            return result

# asyncio.run(async_web_handler())
```

Best Practices {#best-practices}

1. Always Handle Exceptions Properly

python

```python
@contextmanager
def robust_context():
    """
    USE CASE: Production-ready context manager with proper error handling
    WHEN: Building reliable systems that need graceful error handling
    WHERE: Production applications, critical systems, user-facing services
    REAL-WORLD SCENARIO: Database connection management in a web application
    """
    resource = None
    try:
        resource = acquire_resource()
        yield resource
    except DatabaseError as e:
        # Handle specific database errors
        log_database_error(e)
        raise  # Re-raise to let caller handle
    except Exception as e:
        # Handle unexpected errors
        log_unexpected_error(e)
        raise  # Always re-raise unless you know what you're doing
    finally:
        # Always clean up, even if exception occurred
        if resource:
            try:
                release_resource(resource)
            except Exception as cleanup_error:
                # Log cleanup errors but don't raise them
                log_cleanup_error(cleanup_error)

# Usage in web application
def handle_user_request():
    with robust_context() as resource:
        return process_user_data(resource)
```

2. Use ExitStack for Multiple Resources

python

```python
def process_multiple_resources(resources):
    """
    USE CASE: Managing multiple resources with guaranteed cleanup
    WHEN: You need to work with multiple resources simultaneously
    WHERE: Batch processing, data pipelines, resource-intensive operations
    REAL-WORLD SCENARIO: Processing multiple data sources in an ETL pipeline
    """
    with ExitStack() as stack:
        # All resources will be cleaned up automatically, even if one fails
        opened_resources = []

        for resource_config in resources:
            try:
                resource = stack.enter_context(resource_manager(resource_config))
                opened_resources.append(resource)
            except ResourceError as e:
                # Log error but continue with other resources
                print(f"Failed to open resource {resource_config}: {e}")

        # Work with all successfully opened resources
        return process_resources(opened_resources)

# Usage in ETL pipeline
def run_etl_pipeline():
    data_sources = [
        {'type': 'database', 'conn_string': 'db1'},
        {'type': 'file', 'path': '/data/file1.csv'},
        {'type': 'api', 'url': 'https://api.example.com/data'}
    ]

    results = process_multiple_resources(data_sources)
    return combine_results(results)
```

3. Prefer contextlib for Simple Cases

python

```python
# Instead of creating a class, use @contextmanager for simple cases
@contextmanager
def simple_lock():
    """
    USE CASE: Simple synchronization without complex state
    WHEN: You need basic resource protection without complex logic
    WHERE: Thread synchronization, simple resource protection
    REAL-WORLD SCENARIO: Protecting access to a shared file or counter
    """
    print("Acquiring lock")
    try:
        yield
    finally:
        print("Releasing lock")

# Use it for simple synchronization
shared_counter = 0

def increment_counter():
    global shared_counter
    with simple_lock():
        # Critical section
        old_value = shared_counter
        shared_counter = old_value + 1
        print(f"Counter: {old_value} -> {shared_counter}")

# However, for complex cases, prefer classes:
class ComplexResourceManager:
    """
    USE CASE: Complex resource management with state
    WHEN: Resource management involves complex state or multiple operations
    WHERE: Connection pools, complex protocols, stateful resources
    """
    def __init__(self, config):
        self.config = config
        self.connection_pool = None
        self.active_connections = 0

    def __enter__(self):
        self.connection_pool = create_connection_pool(self.config)
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.connection_pool:
            self.connection_pool.close_all()
```

```
            self.connection_pool.close_all()
            self.log_final_stats()

    def get_connection(self):
        self.active_connections += 1
        return self.connection_pool.get_connection()
```

4. Document Context Manager Behavior

python

```python
@contextmanager
def documented_context(resource_name, timeout=30):
    """
    Context manager for managing a named resource with timeout.

    USE CASE: Well-documented resource management for team development
    WHEN: Building reusable components for other developers
    WHERE: Libraries, shared utilities, team codebases

    Args:
        resource_name (str): Name of the resource to manage
        timeout (int): Maximum time to wait for resource acquisition

    Yields:
        Resource: The acquired resource handle

    Raises:
        ResourceError: If resource cannot be acquired within timeout
        TimeoutError: If timeout is exceeded

    Example:
        >>> with documented_context("database_connection") as resource:
        ...     result = resource.query("SELECT * FROM users")
        ...     # Resource is automatically released

    Note:
        - Resource is automatically released even if an exception occurs
        - Timeout applies only to acquisition, not to operations
        - Multiple contexts can be nested safely
    """
    start_time = time.time()
    resource = None

    try:
        resource = acquire_resource(resource_name, timeout)
        if time.time() - start_time > timeout:
            raise TimeoutError(f"Resource acquisition timed out after {timeout}s")
        yield resource
    except ResourceError as e:
        logger.error(f"Resource error for {resource_name}: {e}")
        raise
    finally:
        if resource:
            release_resource(resource)
            elapsed = time.time() - start_time
```

```
        elapsed = time.time() - start_time
        logger.info(f"Resource {resource_name} used for {elapsed:.2f}s")
```

5. Consider Performance Implications

python

```python
# For frequently used context managers, consider caching
class CachedResourceManager:
    """
    USE CASE: High-performance resource management with caching
    WHEN: Context manager is used frequently and resource creation is expensive
    WHERE: High-throughput systems, performance-critical applications
    REAL-WORLD SCENARIO: Database connection pooling, compiled regex caching
    """
    def __init__(self):
        self._resource_cache = {}
        self._lock = threading.Lock()

    @contextmanager
    def cached_context(self, resource_key, create_func):
        with self._lock:
            if resource_key not in self._resource_cache:
                self._resource_cache[resource_key] = create_func()

        resource = self._resource_cache[resource_key]
        try:
            yield resource
        finally:
            # Don't release cached resources
            pass

    def clear_cache(self):
        """Call this during application shutdown"""
        with self._lock:
            for resource in self._resource_cache.values():
                try:
                    resource.close()
                except Exception:
                    pass
            self._resource_cache.clear()

# Usage in high-performance scenario
resource_manager = CachedResourceManager()

def high_frequency_operation(data):
    with resource_manager.cached_context("db_conn", create_db_connection) as conn:
        return process_data(conn, data)

# For lightweight operations, prefer simple context managers
@contextmanager
def lightweight_context():
```

```
def lightweight_context():
    """
    USE CASE: Simple, lightweight resource management
    WHEN: Resource creation/cleanup is fast and caching isn't needed
    WHERE: Simple file operations, temporary state changes
    """
    setup_lightweight_resource()
    try:
        yield
    finally:
        cleanup_lightweight_resource()
```

6. Use suppress Judiciously

python

```python
# Good: Suppress specific, expected exceptions
def safe_file_cleanup():
    """
    USE CASE: Graceful cleanup that handles expected failures
    WHEN: Cleanup operations that might fail due to external factors
    WHERE: Deployment scripts, cleanup routines, maintenance operations
    """
    temp_files = get_temp_files()

    for temp_file in temp_files:
        with suppress(FileNotFoundError):
            # File might have been deleted by another process
            os.remove(temp_file)

        with suppress(PermissionError):
            # File might be locked by another process
            os.remove(temp_file)

# Good: Suppress specific exceptions in optional operations
def optional_feature_cleanup():
    """
    USE CASE: Optional cleanup that shouldn't break the main flow
    WHEN: Non-critical operations that might fail
    WHERE: Plugin systems, optional features, degraded mode operations
    """
    with suppress(ImportError):
        # Optional dependency might not be installed
        import optional_module
        optional_module.cleanup()

    with suppress(ConnectionError):
        # Optional external service might be down
        notify_external_service()

# Bad: Don't suppress broad exceptions
def bad_suppression_example():
    """
    ANTI-PATTERN: Never suppress broad exceptions
    WHY: Hides bugs and makes debugging impossible
    """
    # DON'T DO THIS:
    # with suppress(Exception):  # Too broad!
    #     risky_operation()

    # DO THIS INSTEAD:
```

```
# DO THIS INSTEAD:
try:
    risky_operation()
except SpecificError as e:
    log_error(e)
    handle_specific_error(e)
except AnotherSpecificError as e:
    log_error(e)
    handle_another_error(e)
```

7. Combine Context Managers Effectively

python

```python
def comprehensive_processing(filename):
    """
    USE CASE: Complex operations requiring multiple context managers
    WHEN: Operations need multiple resources and monitoring
    WHERE: Data processing pipelines, complex batch operations
    REAL-WORLD SCENARIO: Processing large files with monitoring, logging, and cleanup
    """
    with ExitStack() as stack:
        # File handling
        f = stack.enter_context(open(filename, 'r'))

        # Performance monitoring
        timer_ctx = stack.enter_context(timer("File processing"))

        # Temporary workspace
        temp_dir = stack.enter_context(temporary_directory())

        # Output capture for logging
        log_buffer = io.StringIO()
        stack.enter_context(redirect_stdout(log_buffer))

        # Database transaction
        db_conn = stack.enter_context(database_connection('processing_db'))

        # All cleanup happens automatically, even if any step fails
        result = process_file_content(f, temp_dir, db_conn)

        # Log the captured output
        captured_output = log_buffer.getvalue()
        if captured_output:
            logger.info(f"Processing output: {captured_output}")

        return result

# Usage in enterprise data processing
def process_daily_reports(report_files):
    """
    REAL-WORLD SCENARIO: Enterprise batch processing with comprehensive error handling
    """
    results = []

    for report_file in report_files:
        try:
            result = comprehensive_processing(report_file)
```

```
            results.append(result)
        except Exception as e:
            logger.error(f"Failed to process {report_file}: {e}")
            # Continue with other files
            continue

    return results

# Advanced pattern: Context manager composition
@contextmanager
def enterprise_processing_context(config):
    """
    USE CASE: Reusable enterprise processing environment
    WHEN: Multiple operations need the same complex setup
    WHERE: Enterprise applications, microservices, batch processing
    """
    with ExitStack() as stack:
        # Set up all required resources
        db = stack.enter_context(database_connection(config.db_url))
        cache = stack.enter_context(redis_connection(config.redis_url))
        logger = stack.enter_context(structured_logger(config.log_config))
        metrics = stack.enter_context(metrics_collector(config.metrics_config))

        # Create processing environment
        env = ProcessingEnvironment(
            database=db,
            cache=cache,
            logger=logger,
            metrics=metrics
        )

        try:
            yield env
        except Exception as e:
            env.logger.error(f"Processing failed: {e}")
            env.metrics.increment('processing_failures')
            raise
        else:
            env.metrics.increment('processing_successes')

# Usage
def run_enterprise_job(job_config):
    with enterprise_processing_context(job_config) as env:
        return process_business_logic(env)
```

This comprehensive tutorial covers the major features and patterns of the `contextlib` module. The key takeaway is that context managers provide a clean, reliable way to manage resources and ensure proper

cleanup, while `contextlib` makes creating them much simpler than implementing the context manager protocol manually.